Supporting data-driven I/O on GPUs using GPUfs

Sagi Shahar
Technion - Israel Institute of Technology
sagi@tx.technion.ac.il

Mark Silberstein
Technion - Israel Institute of Technology
mark@ee.technion.ac.il

Abstract

Using discrete GPUs for processing very large datasets is challenging, in particular when an algorithm exhibit unpredictable, data-driven access patterns. In this paper we investigate the utility of GPUfs, a library that provides direct access to files from GPU programs, to implement such algorithms. We analyze the system's bottlenecks, and suggest several modifications to the GPUfs design, including new concurrent hash table for the buffer cache and a highly parallel memory allocator. We also show that by implementing the workload in a warp-centric manner we can improve the performance even further. We evaluate our changes by implementing a real image processing application which creates collages from a dataset of 10 Million images. The enhanced GPUfs design improves the application performance by $5.6\times$ on average over the original GPUfs, and outperforms both 12-core parallel CPU which uses the AVX instruction set, and a standard CUDA-based GPU implementation by up to $2.5\times$ and $3\times$ respectively, while significantly enhancing system programmability and simplifying the application design and implementation.

Categories and Subject Descriptors. D.4.7 [*Operating Systems*]: Organization and Design; I.3.1 [*Hardware Architecture*]: Graphics processors

Keywords. Operating Systems, GPGPUs, File Systems

1. Introduction

Discrete GPUs are commonly used for speeding up a variety of data processing applications. However, accelerating computations on large data sets which exceed GPU physical memory is a challenge. In fact, many algorithms that are known to be highly efficient on GPUs for small data sets, such as kNN search [9], do not scale to real-world data.

SYSTOR '16, June 6–8, 2016, Haifa, Israel.

http://dx.doi.org/10.1145/2928275.2928276

Computations on large data sets necessarily involve file accesses, but current GPUs cannot access a host file system directly because they lack file system access support. Therefore, an application developer needs to coordinate GPU accesses to secondary storage via explicit application-level management code running on a CPU. This code performs file accesses on GPU's behalf and manages low level data transfers to/from GPU memory. Furthermore, all the data that a GPU may need must be resident in the GPU memory prior to computations, and it is the responsibility of a GPU developer to ensure that this is the case. As a result, all the potential GPU accesses to data must be known *before* the GPU execution starts. This requirement impedes the use of GPUs to run data processing algorithms with irregular data access pattern on large datasets.

Recently, GPUfs [10] introduced file system (FS) support for GPUs. By providing a standard FS API to GPU code, GPUfs enables processing of large datasets by reading files from a running GPU kernel on demand, thereby eliminating the need to know all future GPU data accesses in advance, and obviating CPU-side data management code.

These features make GPUfs ideal for implementing applications with input-dependent data accesses, like image collage. This application takes a single image as an input, breaks it into blocks, and replaces each block with a visually similar image from a large dataset. To quickly find the matching image from a multi-million image dataset, the algorithm employs a Locality Sensitive Hashing (LSH) [11] heuristic, which enables to narrow down the search to only a few images. In particular, the algorithm computes the LSH keys of each block in the input image which in turn identifies a subset of images in the dataset that are then exhaustively searched through. Therefore, the accesses to the dataset are input-dependent and cannot be predicted without computing the LSH first. The collage application represent a large family of LSH-based algorithms used in many areas dealing with large datasets, such as image similarity search [1], image classification [9] or related tweet search [14].

This work examine the applicability of GPUfs to implementing large data-set processing applications with unpredictable data access pattern. Such applications stress various parts of GPUfs, revealing several significant concurrency bottlenecks in the original design. We identify the main

issues, and implement and evaluate the improved design which achieves significant *application* performance boost.

Performance optimization is an inherently iterative process because solving one source of slowdown may potentially expose a number of others, previously hidden scalability bottlenecks. We describe this process in detail. First we look at the traditional read-optimized implementation of the buffer cache which uses radix trees. This implementation fails to scale to support highly concurrent mixed read-/write workloads, induced on the buffer cache by the data-driven access pattern. Yet, replacing the radix tree with a highly concurrent hash map exposes the GPUfs dynamic memory allocator as the next target for optimization. We build a novel, lock-free concurrent allocator which uses multiple rings to eliminate contention among cores. While working well in isolation, it reveals that GPUfs fails to utilize the PCIe bandwidth between CPU and GPU due to the lack of appropriate support for small page sizes, essential to support data accesses with low spatial locality. We introduce opportunistic batching of memory transfers from the host, and show that it *benefits* from higher effective concurrency enabled by earlier optimizations. Interestingly, integrating all the optimized data structures and mechanisms into GPUfs poses a number of additional challenges that we solve and describe in this paper.

We evaluate the performance of the enhanced GPUfs layer by implementing the image collage application. We show that the new version achieves an average of $5.6\times$ speedup over the original GPUfs. Moreover it is up to $2.5\times$ faster than a 12-core CPU implementation using Intel's Threading Building Blocks and the AVX instruction set, and up to $3\times$ faster than a GPU implementation without GPUfs.

In summary, the paper makes the following contributions.

1. We provide an in-depth analysis of the GPUfs scalability bottlenecks in applications with a complex data-driven access pattern.
2. We introduce a new concurrent GPU hash map, a concurrent memory allocator and an opportunistic host-device batching mechanism, and integrate them into the GPUfs
3. We show significant performance improvements on realistic workloads over CPU and native GPU implementations, as well as over the original GPUfs version.

The rest of the paper is structured as follows. Section 2 provides an overview on GPUs, the GPUfs system, the LSH algorithm, and the image collage application, which is based on the LSH programming scheme. Section 3 analyzes the impact of the LSH input-dependent memory access pattern on the system performance, highlighting the main GPUfs bottlenecks. We then describe our optimizations of the original GPUfs design in Section 4, and additional modifications required to support the full set of GPUfs API in Section 5. We present our results in Section 6, discuss related work in Section 7 and conclude in section 8.

2. Background

We provide a brief description of GPU programming principles and GPUfs system.

2.1 GPUs

We briefly outline the basic concepts of the GPU architecture and programming model, focusing on discrete GPUs, using NVIDIA CUDA terminology; more details about CUDA and the GPU model can be found in [6].

Execution hierarchy. GPUs are parallel processors that run thousands of threads. Threads are grouped into *warps*, warps are grouped into threadblocks, and multiple threadblocks form a GPU kernel which is invoked on the GPU by the CPU. A warp is a set of 32 threads that are executed in lockstep, i.e., at a given step all the threads in the warp execute the same instruction. Threads in a threadblock are invoked on the same core called Streaming Multiprocessor (SM) and may communicate and synchronize efficiently.

Memory hierarchy. Each thread has a set of dedicated private registers. Registers are the fastest type of memory. Threads in the same threadblock share a fast threadblock-private on-die scratchpad called *shared* memory. All threads in the kernel share *global* GPU memory. Global memory provides about an order of magnitude lower bandwidth than shared memory. Accesses to global memory are cached in a two-level hardware cache. The GPU can also access the CPU's memory over the PCIe bus, using pinned pages. Such access has many limitations such as lack of cache coherency and atomic operations, incur high latencies, and is an order of magnitude slower than the GPU's internal global memory.

Inter-thread communication. Threads in the kernel may communicate via global memory. Threads in the same threadblock may also communicate via shared memory, and synchronize by using efficient hardware barriers.

2.2 GPUfs

GPUfs [10] is a file system layer which exposes a POSIX-like file system API to GPU applications.

The high level design is presented in Figure1. GPUfs library is linked to the GPU kernel. It provide the basic file operations such as `gopen`, `gclose`, `gread` and `gwrite`, as well as `gmmap`, `gfsync`, `gftruncate`, etc. All these calls are blocking calls, similarly to the standard CPU OS system calls.

In the original GPUfs implementation, these calls have threadblock level semantics, meaning that all the threads in the threadblock are required to issue the call together, using the same arguments. This semantics is easier to implement and use, however higher performance can be achieved by a finer-grained warp-level semantics, as we show in Section 4.

In addition, GPUfs manages a buffer cache in GPU memory in order to minimize redundant accesses to the host file system. The buffer cache itself is managed in pages of the same size, configured at system initialization. The man-

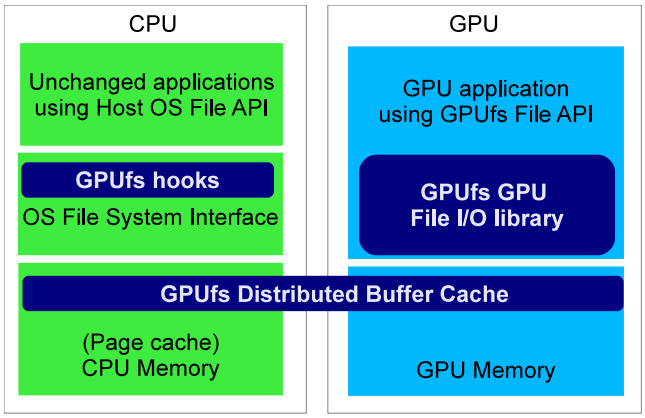

Figure 1. GPUfs Architecture (reprinted from [10])

agement of these pages is achieved using a radix tree per opened files. Each radix tree allows lock-free reads but require global locking for updates.

If a requested page is not found in GPU memory, GPUfs accesses the host file system by issuing a Remote Procedure Call that is handled by the CPU. When there's no more available space in the buffer cache, old pages are evicted, base on Least Recently Used policy. Note that since all pages in the buffer cache has backing store in the file system, the buffer cache does not need to handle a swapping area.

The GPUfs system supports the close-to-open consistency model, which is also used in the NFS file system [3]. Meaning that files that are re-opened in the GPU, after being touched and closed in the CPU, need to be invalidated and re-read into the GPU buffer cache in order to reflect the updates done on the CPU.

2.3 The Locality Sensitive Hash algorithm

Locality Sensitive Hash-based algorithms is a family of algorithms used mainly for fast retrieval of high dimensional data from large datasets. These algorithms use special *locality-preserving* hash functions, which map *neighboring* objects in high dimensional space into approximately neighboring objects in the low dimensional space. In other words, neighboring objects in the data set are likely to have the same hash key and therefore placed in the same *bucket* in a hash table. This property makes LSH particular useful for a variety of applications that require sub-linear-time data retrieval from large datasets, like image similarity search [1], related tweets search [14] and image classification [9].

The use of LSH in these applications follows a common structure as we describe below.

First, the dataset is reorganized off-line, such that each object is placed in multiple hash tables, each with its own LSH function.

The online part, which we implement in this paper, is detailed in Algorithm 1.

Algorithm 1 LSH based Approximate Nearest Neighbor

```
for each query do                          ▷ Can be done in parallel
    Extract features
    for i ← 1, numKeys do
        k ← calculate LSH key
        for each item in bucket[k] do
            Insert to shortList            ▷ Unique list
        end for
    end for
    for each item in shortList do
        Read from file
        Calculate distance
        Update minimum
    end for
end for
```

1. Perform feature extraction for the query and compute its LSH hash keys. This stage executes independently for each query, requires no access to the actual dataset, and its compute intensity can vary depending on the feature extraction method.
2. Fetch candidate objects from the dataset by retrieving the buckets corresponding to the LSH keys computed in (1). Merge all the retrieved objects into a shortlist while removing duplicates (different buckets may contain the same object).
3. Choose the closest object(s) to the query by performing brute-force search in the shortlist. This is the most compute intensive part of the algorithm since the list may contain several dozens of entries.

The second stage of the algorithm accesses the dataset based on the keys computed in the first stage. Therefore, the actual data accesses are input-dependent and cannot be predicted in advance. Moreover, these algorithms are applied to very large datasets that do not fit even into CPU memory, and certainly exceed the even more limited GPU memory, making prefetching of the complete data set into CPU or GPU memory is impossible.

2.4 Image collage application

In this paper we use an approximate nearest neighbor search based on LSH as a building block for an image collage application.

The application accepts an image from the user. For each 32×32 pixels block in the image, we search for a *similar* image inside a 10 Million images dataset. The dataset itself holds the precomputed Red-Green-Black (RGB) histograms of the input images in a continuous array, while each histogram location is 4KB aligned.

We create 32 hash tables (using different LSH functions), where each hash table bucket contains the file offsets of images corresponding to that bucket. The application is implemented as follows. In the first stage we compute the LSH keys of each of the 32×32 pixels blocks in the input im-

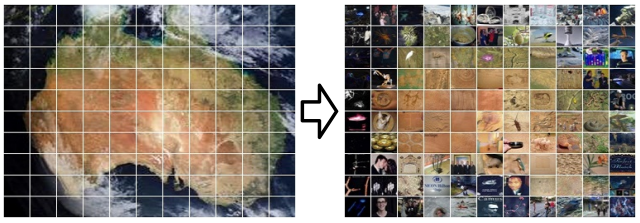

Figure 2. Collage example

age, based on the block color histogram. Then for each of those keys, we gather the locations of candidate images in the dataset. In the last stage we read the candidates from disk and perform the short list comparison to find the best match. An example image created by the collage application is presented in Figure 2.

3. GPUfs performance analysis

In this section we analyze the performance of GPUfs on a complex workload by implementing the image collage application described in Section 2.4.

This application is among the most appealing use cases for GPUfs. Yet, it poses many challenges to the GPUfs system due to its data-driven accesses to files reading small blocks of data (3KB at a time) with low spatial locality. In particular, it induces many concurrent accesses to different pages in the GPUfs buffer cache, resulting in a mixed read-/write workload on the internal buffer cache data structures. Further, it requires frequent transfers of small data chunks into GPU memory, stressing the data transfer subsystem running on the host.

As we show shortly, the original GPUfs implementation turns out to be highly inefficient for this workload, setting the stage for a thorough investigation of the bottlenecks and the consequent changes in the design to eliminate them, which we describe in Section 4.

Implementation. We evaluate three versions of the collage algorithm.

- **CPU**: Multithreaded CPU version implemented using Intel's Threading Building Blocks (TBB) [5]. The computation portions of the application are optimized using the AVX instruction set [12].
- **CPU-GPU**: Traditional GPU-accelerated version which uses two GPU kernels for Step 1 and Step 3 of Algorithm 1 respectively, and uses CPU to create the shortlist and read data from files in Step 2.
- **GPUfs**: GPUfs version, implemented as a single kernel executing all the stages of the algorithm, including Step 2 for which it uses GPUfs to accesses file system.

Both the CPU and GPUfs versions follow a straightforward approach to access data from files. Specifically, after computing the LSH values and finding the buckets with the indices of the candidate images in Stage 2, they fetch the images from the dataset in Stage 3 by mapping the relevant parts of the file using `mmap` (`gmmap` in GPUfs). Importantly, If an image is used by two different blocks, only the first access to the image will result in disk access, while the rest will be served from the buffer cache. Thus, these implementations take advantage of the file system layer to eliminate redundant reads from disk.

In contrast, the CPU-GPU implementation must explicitly eliminate such redundant accesses. This is because even though the CPU buffer cache does prevent redundant disk accesses, the CPU must transfer the images to GPU memory for performing Step 3 on the GPU. Thus, if the same image appears twice in the shortlists of different blocks, it will be read once from the disk, but transferred twice to the GPU.

In our implementation, the CPU first filters out duplicate images by using a hash table, then reads the images into a staging area in CPU memory [1], builds a list of memory offsets to each image in the buffer, and transfer the images with the list to the GPU. One particular limitation of this implementation is that it only works as long as all the images in the shortlists fit into GPU memory. The GPUfs-based implementation does not suffer from such limitation.

Dataset. We evaluate all the three implementations by generating collages from the Tiny Images dataset [2]. We use a subset of this dataset with 10 million images. We store the histograms of all the images in a continuous array, 39GB of raw data in total, including padding for 4KB data alignment.

Hardware/Software configuration. We run on a Super-Micro server featuring 2×6-core Intel i7-4960X CPUs at 3.6GHz with 15MB L3 cache per CPU, and an NVIDIA GeForce GTX TITAN GPU with 6 GB of GDDR5 memory. We run Ubuntu Linux kernel 3.13.0-32, with CUDA SDK 7.0 and NVIDIA GPU driver 346.29. GPUfs is configured to use 4K pages in its buffer cache, in order to match the sizes of the histograms. We store the data in CPU memory using RAMfs to avoid slow disk accesses and analyze the performance of the software implementation. We run each test ten times, clearing the GPU buffer cache between runs, and use the first 3 runs as a warm up. We report an average of the last 7 iterations.

Results.

Figure 3 shows the results of running the collage on several images of increasing sizes ranging from 512×384 (192 blocks) to 4096×3072 (12288 blocks). The CPU version outperforms both the CPU-GPU nor GPUfs implementations. Moreover, the GPUfs implementation is the slowest across all the image sizes. We further investigate this problem and resolve it in the next section.

[1] CPU memory used for data transfers to/from the GPU must be pinned, so we cannot copy directly from the CPU buffer cache.

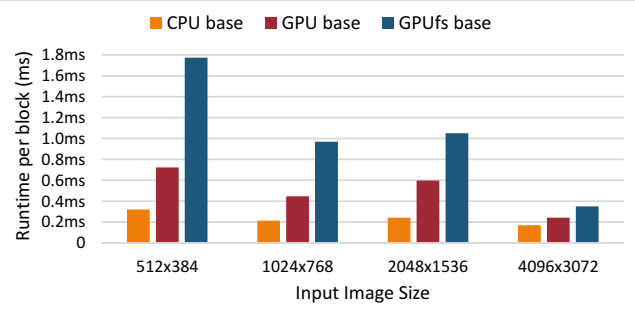

Figure 3. Runtime of the image collage implementations, normalized per image block. Lower is better

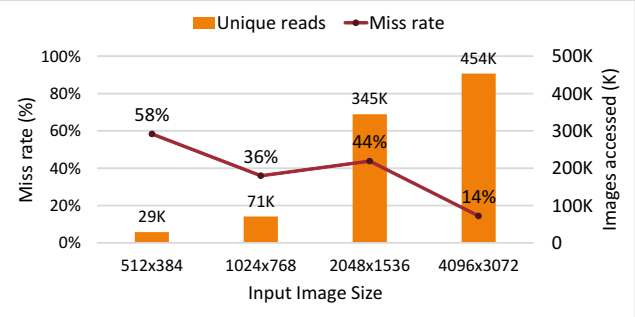

Figure 4. Buffer cache miss rate for images of different sizes. The right axis shows the number of unique pages accessed. Some images experience lower miss rate because neighboring blocks in the image are mapped to the same LSH bucket due to smoothness of natural images

Image size	Access time (ms)	Spinlock time (ms)
512x384	291 (96%)	286 (95%)
1024x768	665 (96%)	654 (95%)
2048x1536	3011 (96%)	2950 (94%)
4096x3072	3827 (91%)	3718 (89%)

Table 1. Aggregate radix-tree based buffer cache access time. The number in parentheses is the percentage of the total running time, lower is better

4. GPUfs Analysis & Optimizations

In this section we dive deeper into the GPUfs performance, focusing on its three major components: the buffer cache, the memory allocator, and the CPU-GPU transfer mechanism.

As we show, each of these components does not scale under high concurrency and becomes a performance bottleneck. However, poor scaling of one component may potentially hide the scalability limitations of the others. We therefore optimize one component at a time, and iteratively eliminate all these major bottlenecks.

4.1 Radix tree bottleneck

GPUfs maintains all the buffer cache contents in a set of radix trees, one tree per file.

In order to evaluate the overhead of the radix tree we time the radix tree traversals while accessing the buffer cache. We use `clock()` GPU intrinsics which samples internal GPU clock.

The first column in Table 1 shows that the radix tree search time comprises up to 96% of the total runtime. Further analysis shows that the implementation suffers from heavy contention, spending more than 97% of the radix tree search time on a global spinlock. As a result, accesses to files from different threadblocks are serialized.

These performance bottlenecks have not been observed in the workloads with which GPUfs was evaluated earlier, such as matrix product and image search, because they have high data reuse (over 99%) and result in predominantly read-only access pattern on the radix tree. The original radix tree design in GPUfs has been optimized for read-only access and uses no locks at all for such accesses.

However, in the image collage we consider here, the buffer cache miss rate is high, in particular for smaller images, and therefore the radix tree experiences a mixed read-/write workload. Figure 4 shows that up to 58% of the file accesses result in buffer cache misses. Those accesses modify the radix tree, acquiring a global spinlock first, which leads to major contention and slowdown.

Solution: concurrent hash table. We replace the per-file radix tree design with a single concurrent hash table. The table resolves a tuple `[file ID, file offset]` to the buffer cache page holding file data in GPU memory. The capacity of the hash table is set to be 16 times the number of pages in the buffer cache. This configuration provides a reasonable balances between the collision probability (theoretically 3% for fully occupied buffer cache) and the memory overhead (5% of the buffer page size). We implement fine-grain locking per bucket for insertion and lock-free reads.

We also considered a concurrent lazy list [15] as an alternative for the lock-per-bucket design, but decided against it. In the lazy list implementation each insertion or removal requires two locks, which improves concurrency for hash tables with high collision rate. In our case, however, these locks are likely to lock the entire bucket since the average size of each bucket is less than 2. In our implementation we require only a single lock per bucket, while providing similar level of concurrency.

We integrate the new hash table-based buffer cache with GPUfs and evaluate the performance under high contention by eliminating CPU-GPU data transfer overhead (we replace the CPU-GPU data transfers with empty stubs). This results in about 350K insertions per seconds when running the collage application.

Table 2 confirms that the new hash table implementation is much faster. The buffer cache access time is reduced

Image size	Access time (ms)	Allocation time (ms)
512x384	48 (84%)	43 (76%)
1024x768	124 (81%)	114 (74%)
2048x1536	696 (88%)	615 (78%)
4096x3072	777 (58%)	616 (46%)

Table 2. Aggregate hash table-based buffer cache search time and memory allocation time using old memory allocator. The number in parentheses is the percentage of the total running time. The evaluation is performed without CPU-to-GPU data transfers

Image size	Access time (ms)
512x384	3 (21%)
1024x768	8 (15%)
2048x1536	89 (37%)
4096x3072	161 (19%)

Table 3. Aggregate buffer cache access time (not including CPU-GPU data transfers) in the implementation that combines concurrent hash table and ring-buffer memory allocator. The number in parentheses is the percentage of the total running time

by $5\times$ on average compared to the radix tree-based implementation performance shown in Table 1 even under much higher contention.

4.2 Memory allocator bottleneck

Despite the improvement, we observe that buffer cache accesses time still occupies a significant portion of the total runtime. Further investigation reveals a new, previously hidden, bottleneck. As we see in Table 2 the access time is dominated by the memory allocator used to allocate new buffer cache pages. This is, in fact, not surprising because the GPUfs memory allocator uses a simple free list with a single global lock. The radix tree bottleneck masks this obvious scalability bottleneck, however the memory allocator becomes a hotspot once the new efficient concurrent hash table implementation is in place.

Solution: ring-buffer allocator. We implement a memory allocator using a ring buffer. Each element points to a pre-allocated page in the GPU memory. Pages are thus allocated by updating the head pointer via a simple atomic operation without using locks.

The deallocation is only performed as a result of page eviction from the buffer cache – pages are not returned to the pool of free pages otherwise. Thus, the system does not deallocate specific page, rather its goal is to free some space in the buffer cache. We discuss the page eviction in Section 4.4.

The new allocator reduces the overall buffer cache access overhead by an average of $10\times$, even when dealing with 530K insertions per second that can be delivered by the new system.

The end-to-end performance of the new buffer cache implementation including the new memory allocator and concurrent hash table is presented in Table 3.

4.3 Optimizing CPU-GPU transfers

The performance of memory transfers between the CPU and the GPU over the PCIe bus is particularly critical in I/O-bound applications, such as image collage.

We estimate the data transfer by using a standard effective bandwidth metric, defined as the ratio between the total amount of data transferred from the CPU to the GPU, and the aggregate application running time. This metric is useful since it reflects both the actual throughput per request, and the system ability to overlap computation and I/O. In this work we improve the effective bandwidth by optimizing both these factors as we show in this section and in Section 4.3.2.

Background: data transfers in GPUfs. GPUfs needs to transfer data in two cases: upon reading it from the file, and upon writing it back as a result of page eviction or file write back. In this paper we primarily focus on the read data path. When `gread` request cannot be served from the buffer cache, the GPU-side library issues the request to the CPU-side Remote Procedure Call (RPC) daemon to transfer the file contents directly into the buffer cache page. This transfer is performed as part of the `gread` call, hence the call is blocking. The CPU-side daemon transfers one page at a time.

GPUfs performance with 4K pages. We run the image collage workload using the modified GPUfs with the enhanced buffer cache as we described above. We store the file contents in CPU RAM using RAMfs, and compute the effective bandwidth. We measure very low effective bandwidth of 0.31GB/s on average across all the input images. The obvious conclusion is that the PCIe is underutilized.

Increased page size does not work. One obvious solution is to increase the system page size in GPUfs. This solution, in fact, has been shown to work well in GPUfs when used with simple streaming workloads, or workloads with high spatial locality. In such workloads large pages are efficient because all the data in the page are eventually used by the program.

In workloads with low spatial locality and small reads, like in image collage where each threadblock reads 3KB per histogram, the page size increase leads to redundant data transfers and therefore harms performance. The following experiment confirms this observation.

We run the image collage multiple times, each with different page size ranging from 4KB to 64KB. For each run we measure the relative amount of redundant data transfers. Specifically, we compute the ratio between the total amount of data transferred to the GPU from the CPU, and the amount of data that the algorithm actually needs for computations.

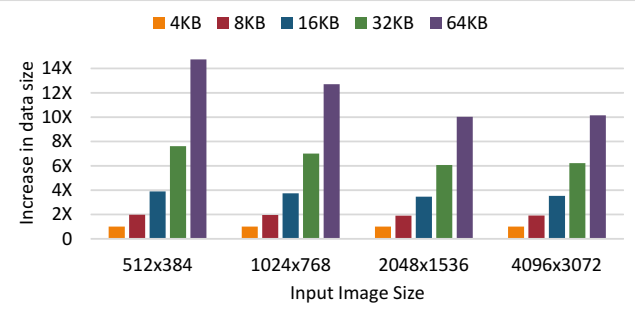

Figure 5. The amount of redundant data transferred for different buffer cache page sizes and different inputs. For example, 2 means that there was twice more data transferred than used

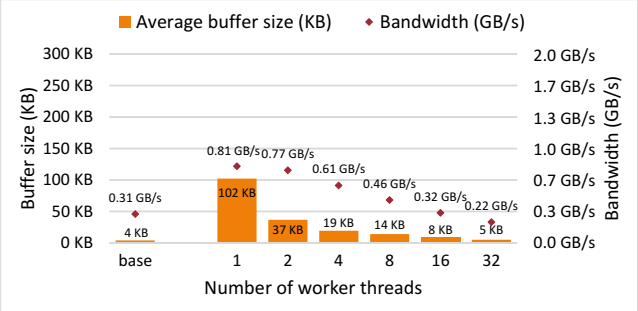

Figure 6. Effective bandwidth and average transfer size with batching (threadblock-level API calls)

For applications which transfer no redundant data, this ratio is 1.

As Figure 5 shows, running the image collage with larger pages results in large amount of redundant data transfers. Doubling the page size from 4KB to 8KB almost doubles the amount of transfered data, half of which remains unused, wasting the CPU-GPU memory bandwidth. Larger pages only exacerbate the problem.

4.3.1 Solution: opportunistic I/O Batching

The change involves two parts: batching of requests on the CPU, and dispatching the file contents to the respective pages in the buffer cache in the GPU.

The first part is performed by the CPU. The GPU's RPC requests are accumulated in the RPC request queue shared between the CPU and the GPU. The CPU reads the data from files into the stage area and transfers it as a single continuous buffer into the GPU staging area.

The second part is performed by the GPU. After the data transfer completes, the CPU notifies each GPU threadblock which requested it. Then the threadblock copies the contents from the GPU staging area into the respective GPU buffer cache page.

Batch size and batching delay. Usually, batching involves delaying certain requests in order to accumulate more data from multiple transfer requests. In our case, delaying the requests on the CPU may actually hurt the application performance. This is because GPUfs `gread` calls are blocking and use busy wait in GPU, wasting GPU resources due to the lack of an I/O preemption mechanism in GPUs. Therefore, we employ *opportunistic* batching, which relies on the high *rate* of `gread` requests from the GPU, and naturally enables accumulating several requests while the previous batch is being transferred. As a result, the number of pages per batch depends on the GPU request rate.

Results. We evaluate the batching mechanism by running the collage application using the largest image as its input. The results are shown in Figure 6. We run several experiments, with different number of *worker* threads on the CPU that are responsible for handling data transfer requests.

We compare both the average transfer size per transfer, and the effective transfer bandwidth experienced by the application. Increasing the number of worker threads leads to lower effective bandwidth. This is because the GPU requests handling rate by multiple threads is higher than the GPU request rate, as is also evident from the reduction in the average transfer size. This workload achieves a maximum effective bandwidth of 0.81GB/s for a single worker thread, improving the original design by 2.6×.

4.3.2 Increasing the request rate

We aim to further improve the effective bandwidth by increasing the GPU read request rate. The original GPUfs provides threadblock-level API, namely, it requires all the threads in a single threadblock to issue the GPUfs API calls at the same point in a program with the same arguments. While the threadblock-level API is easier to use, it limits the maximum number of concurrent requests to the number of actively running threadblocks, which ranges from tens to about a hundred in a typical GPU application running on modern GPUs. We can significantly increase this number by reducing the granularity of the GPUfs API calls from threadblock-level to the level of a single warp.

We therefore implement the GPUfs API with a warp-level granularity. We note that this implementation would be inefficient in the original GPUfs without the buffer cache optimization we described earlier.

In addition, we also modify the collage application in order to use these fine-grain API as follows. As before, one threadblock is allocated to process one block of the input image. However, the shortlist of the dataset images for that block is divided between the threadblock warps. Each warp reads the data from the file using warp-level GPUfs API, and processes it *independently* of the other warps. Since now each warp produces one best match, the final threadblock result is computed by choosing the best among the warp's results.

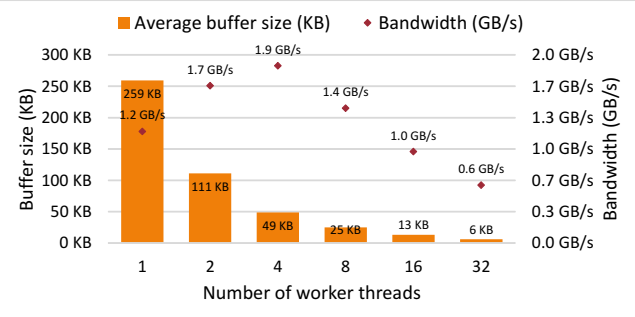

Figure 7. Effective bandwidth and average transfer size with batching of warp level workload

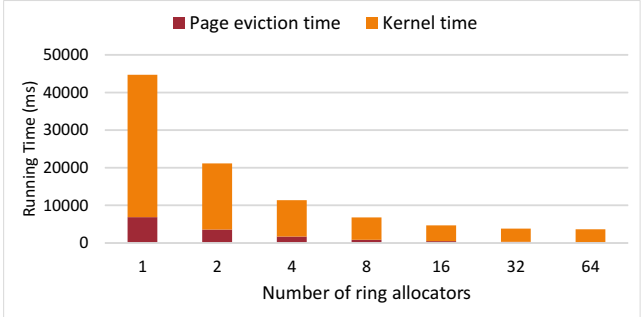

Figure 8. Processing time of a 8192x6144 image while varying the number of memory allocators

The warp-level implementation increases the number of in-flight GPU file read requests by $8\times$.

Results. We evaluate the effective bandwidth of the warp-level implementation and show results in Figure 7. As in Figure 6, we vary the number of worker threads used for data transfer on the CPU. We see that four workers are necessary to achieve the maximum bandwidth of $1.9GB/s$. This is more than a $2\times$ increase in the highest bandwidth compared to the batching results with the threadblock-level design.

4.4 Resolving page eviction bottleneck

Our collage application supports large images, requiring more data than fit into the buffer cache, or in the GPU memory. For example, processing an image of size 8192x6144 requires reading 1.3 million unique pages (4.9GB). Since the buffer cache is configured to 2GB, processing this image necessarily results in page eviction from the buffer cache. We note that the buffer cache is essential for processing this image, because about 89% of file accesses read the data that has already been accessed before (and would be read from the buffer cache if it were large enough to hold the entire working set).

We evaluate the system performance under page eviction, and run the warp level implementation of image collage on this large image. We find that the processing time is as high as 44s, which is more than $45\times$ the processing time for the 4096x3072 image with small working set that fits in the buffer cache.

While investigating the reasons for the slowdown we found that the eviction mechanism is too slow, and eventually the whole GPU stalls while waiting for free pages.

The eviction mechanism traverses the ring buffer of the memory allocator from the tail pointer to find the pages to evict. Ideally, evicting pages from the tail is fast, and implements the least recently allocated eviction policy, which has been used in the original version of GPUfs. Unfortunately, such eviction is not always possible because the page may be pinned, and therefore cannot be evicted. We, therefore, keep searching for the non-pinned page. When found we swap the victim page with the one pointed by the tail pointer in the ring buffer and evict the victim page. This process is protected by a global lock for simplicity, therefore the eviction is performed by a single thread and is slow.

Solution: multiple ring-buffer allocators. To reduce contention when evicting pages we use two techniques. First, we start evicting pages when the ring buffer is nearing its full capacity but there are still pages left for allocation. This allows the allocator to service allocation requests while pages are evicted from the buffer cache.

The second technique is to distribute the available memory between multiple allocators. Each allocator runs independently.When a thread allocates a new page, the allocation is performed by one of the allocators chosen based on the ID of the calling thread.

Figure 8 shows the processing time of a 8192x6144 image while the system uses multiple allocators. The performance improves $12\times$ compared to a single allocator. Further, the percentage of the overall execution time consumed by page eviction drops significantly from 15% for the single allocator to 0.06% when using 64 allocators.

5. Adjusting GPUfs to hash table-based buffer cache

The new buffer cache design that uses one global hash table instead of the radix-tree-per-file design in the original GPUfs introduces a new challenge: it removes the ability to quickly find the set of pages belonging to a certain file. Maintaining this information creates a contention point on the first access to the page, and requires bookkeeping upon page eviction.

We now explain how we modify GPUfs to accommodate this change.

5.1 Write back

GPUfs writes back dirty pages to disk after a file is closed by the GPU to implement close-to-open consistency with the CPU, or after an explicit call to `gfsync`.

In both cases we need to traverse the file's dirty pages so that they can be written to disk.

Solution: multiple writeable pages lists. For every file we maintain several linked lists, each holding a subset of writable pages. The list to which the writable page is added is determined by the thread ID, thus reducing contention when pages are inserted to the writable pages list by multiple threads. Importantly, there is no danger of adding the page twice to the writable page list, because the pages are added to the list on the first insertion to the buffer cache.

This approach is simple and saves the overhead of maintaining the list while accessing read-only files. However, it is too coarse-grained, because it keeps track of all the pages in files opened with write permissions, instead of a more fine-grained management of dirty blocks alone. We leave this optimization for future work.

5.2 File invalidation

GPUfs implements a close-to-open consistency model, in which the file updates are guaranteed to become visible to other processes only when the file is closed and then re-opened again. Implementing this model requires invalidating the cached contents of a file in the GPU memory. In particular, if a file is cached on the GPU, modified by the CPU, and then again accessed by the GPU, the GPU buffer pages caching the file contents must be invalidated.

Solution: file versioning. We assign a version number to each file descriptor and to each page in the system. When a file is opened for the first time, it is assigned a version number. The first time each page is inserted into the buffer cache it is assigned the same version number. On each subsequent access the page version must match the file version, otherwise the page is invalidated and re-fetched from the CPU.

When a file is reopened in the GPU after it has been modified by the CPU its version number is incremented. Thus, the pages are invalidated lazily.

6. Results

We combine all the optimizations and changes, and show the end-to-end performance for two applications. The first is the image collage application described in section 2.4 which was used throughout the performance analysis phase. The second is a brute force, approximate image search, which was used for the evaluation of the original GPUfs system. Our goal has been to show that the optimizations which contribute to the performance of the complicated collage task do not harm the performance of other applications.

6.1 Image collage application

Figure 9 shows the overall performance evaluation for the image collage application for the different implementations.

Note that for the 8192×6144 image, the GPU implementation is unable to run at all since the required data exceeds the physical memory size of the GPU. Given that, when calculating average speedups which involves GPU implementation, we omit results for the largest image.

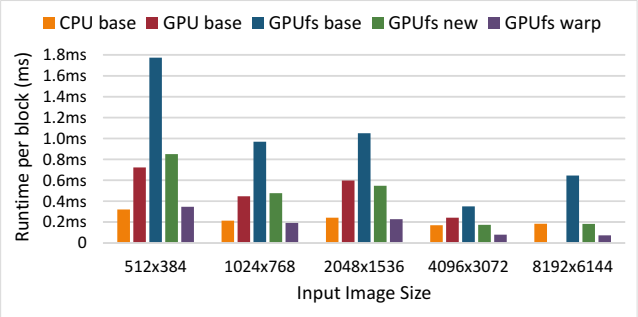

Figure 9. Runtime of the image collage implementations, normalized per image block. Lower is better

The new GPUfs implementation achieves $2\times$ speedup on average over the original GPUfs on the image collage application. When using the warp level workload we can increase the average speedup to $5.6\times$ over the original GPUfs implementation. Importantly, our optimizations of the GPUfs layer allow the warp-level GPUfs-based collage implementation to outperform the 12-core CPU implementations by an average of $1.5\times$ with speedup as high as $2.5\times$ for the largest image. Comparing our implementation to the native CUDA-based GPU implementation, we can see an average of $2.5\times$ with a speedup as high as $3\times$, while using fewer lines of code.

6.2 Approximate image search

The approximate image search application accepts as input a set of query images, and several image datasets. The application then performs brute force search, comparing each query image to the images in each of the datasets until a match is found. The image matching is done using the euclidean distance metric, similarly to the one used in the image collage application.

We run the application using 560 queries and 3 datasets, each with about 10000 images. Each image is represented as a 1K-element vector, same as a 32x32 gray scale image.

Since the application performs brute force search on the datasets, it exhibits sequential data access pattern for dataset accesses, which reduces write-contention on the buffer cache. When data is being read sequentially by multiple threads, most threads will request either the same pages as the other threads, or pages which has already been read by the other threads. For such an access pattern, the number of reads which causes page faults is small.

Exploring the application access pattern further, we can see that since the entire dataset is being read sequentially, the application exhibit perfect spatial locality and therefore increasing the page size will only improve performance. In addition, each dataset which is read into the GPU memory is used for comparison against each of the queries, in our case 560 times, achieving a much higher temporal locality than the one achieved by the image collage application.

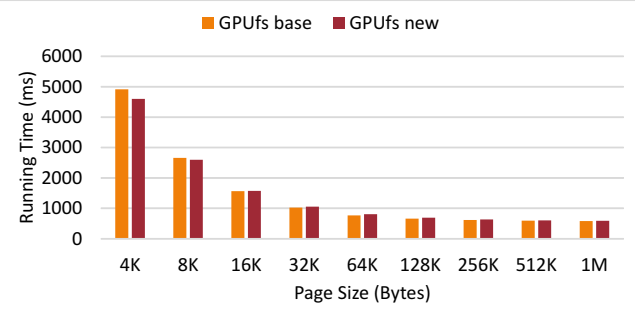

Figure 10. Approximate image search application performance for different page sizes, for both the baseline and new GPUfs implementations

We run the application on both the baseline and the new GPUfs implementations and vary the system's page size for both implementations. We present the results in Figure 10.

As can be seen, both implementations achieves similar performance across all page size, with differences within $\pm 7\%$. Both implementation also shows significant speedup of $8\times$ when increasing the page size from 4KB to 1 MB.

As expected, given the application's access pattern, our changes to the GPUfs system had little effect on the overall performance, neither improving it nor harming it.

7. Related work

The problem of automatic and efficient GPU data management has been addressed in several recent works, however handling input-dependent data accesses from GPU to files is the novel contribution of this paper.

Several approaches have been proposed to handle large datasets. Some of them are application-specific, like Shredder [13] while others, such as PTask [4] require using a special programming model. Both of these approaches use data chunking and multiple kernel invocations to achieve their goal. However, static data chunking requires advanced knowledge of the data that will be accessed at each stage of the algorithm. Therefore, for many structured applications such as dense matrix operations or streaming applications, data chunking works well. Yet, this approach is difficult to apply for applications that exhibit input-dependent data accesses, like the one used in this paper.

Lee at al. [8] propose to predict an application data access pattern by combining static analysis of GPU kernel code, and execution of an inspection kernel. This inspection kernel is a modified version of the original GPU kernel leaving only the code responsible for computing data access locations. This mechanism, however, cannot handle complex data-driven accesses.

NVIDIA recently introduced a Unified Virtual Memory mechanism that allows both CPU and GPU to share data using a unified address space. The memory is allocated on the device and is moved transparently between the GPU and the CPU on demand. Even though this mechanism may be used to store our dataset and let the GPU driver handle the necessary data transfers, it has two major limitation in the context of our application that accesses files. (1) Data access from a CPU to the buffer located in Unified Memory can not be performed while a kernel is running. In particular, in order to access data from a file, a GPU kernel still needs to be broken into several separate kernels, where the file accesses are performed by a CPU between kernel invocations. (2) The maximum size of the Unified Memory buffer is limited because this buffer is allocated on the GPU. Therefore its size cannot exceed the physical memory size of the GPU. This limitation only allows us to work on datasets that can reside inside the GPU memory and is inapplicable to processing large datasets, which is the main focus of this work.

Similarly to the Unified memory concept by NVIDIA, Region-based Virtual Memory for GPUs was proposed by Ji at al. [7]. Here, the buffers are allocated on the host, therefore the allocation size is not limited to the size of GPU physical memory. Further, data can be accessed from the host while the GPU kernel is running. This approach, however, still requires the entire dataset to be copied into a dedicated memory location on the host prior to GPU execution and does not support direct file system access from GPU.

8. Conclusion

In this work we explore the utilization of the GPUfs system to handle the data management requirements of irregular workloads. Such workloads can exhibit unpredictable, fine grain accesses to a large dataset which often can not fit in its entirety inside the GPU physical memory. We examine the LSH-based image collage application as a representative example of such irregular workloads.

While the original implementation is able to support such workloads, the basic design of the GPUfs system is unable to handle the demanding requirements of such workloads efficiently, and fail to reach a higher performance than the CPU implementation.

We propose several modifications, both to the data structures used for the internal management of the GPUfs system, and to the data transfer mechanism between the host and device. By incorporating these modifications we show a significant performance improvement for irregular workloads while maintaining the same performance for previously examined workloads.

Acknowledgements

Mark Silberstein is supported by the Israel Science Foundation (grant No. 1138/14), the National Science Foundation (grant No. CCF-1333594), the Israeli Ministry of Science, and the Israeli Ministry of Economics via HiPer consortium.

References

[1] Aleksandar Stupar, Sebastian Michel and Ralf Schenkel. RankReduce–Processing K-Nearest Neighbor Queries on Top of MapReduce. In *Proceedings of the 8th Workshop on Large-Scale Distributed Systems for Information Retrieval*, pages 13–18, 2010.

[2] Antonio Torralba, Robert Fergus and William T Freeman. 80 Million Tiny Images: A Large Data Set for Nonparametric Object and Scene Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(11):1958–1970, 2008.

[3] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel and Dave Hitz. NFS Version 3: Design and Implementation. In *USENIX Summer*, pages 137–152. Boston, MA, 1994.

[4] Christopher J Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray and Emmett Witchel. PTask: Operating System Abstractions to Manage GPUs as Compute Devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 233–248. ACM, 2011.

[5] Chuck Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

[6] David B Kirk and Wen-mei W Hwu. *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

[7] Feng Ji, Heshan Lin and Xiaosong Ma. RSVM: A Region-Based Software Virtual Memory for GPU. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 269–278. IEEE, 2013.

[8] Janghaeng Lee, Mehrzad Samadi, Scott Mahlke. VAST: The Illusion of a Large Memory Space for GPUs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pages 443–454. ACM, 2014.

[9] Jia Pan and Dinesh Manocha. Fast GPU-based Locality Sensitive Hashing For K-nearest Neighbor Computation. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 211–220. ACM, 2011.

[10] Mark Silberstein, Bryan Ford, Idit Keidar and Emmett Witchel. GPUfs: Integrating a File System with GPUs. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 485–498. ACM, 2013.

[11] Mayur Datar, Nicole Immorlica, Piotr Indyk and Vahab S Mirrokni. Locality-Sensitive Hashing Scheme Based on P-Stable Distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, pages 253–262. ACM, 2004.

[12] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri and Shihjong Kuo. Intel avx: New frontiers in performance improvements and energy efficiency. *Intel white paper*, 2008.

[13] Pramod Bhatotia, Rodrigo Rodrigues and Akshat Verma. Shredder: GPU-Accelerated Incremental Storage and Computation. In *FAST*, page 14, 2012.

[14] Saša Petrović, Miles Osborne and Victor Lavrenko. Streaming First Story Detection with Application to Twitter. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 181–189.

[15] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N Scherer III and Nir Shavit. A Lazy Concurrent List-Based Set Algorithm. In *Principles of Distributed Systems*, pages 3–16. Springer, 2006.